# Modeling Erlang Processes as Petri Nets

Jörgen Brandt
Humboldt-Universität zu Berlin
Germany
brandjoe@informatik.hu-berlin.de

Wolfgang Reisig
Humboldt-Universität zu Berlin
Germany
reisig@informatik.hu-berlin.de

## Abstract

Distributed systems are more important in systems design than ever. Partitioning systems into independent, distributed components has many advantages but also brings about design challenges. The OTP framework addresses such challenges by providing process templates that separate application-dependent from application-specific logic. This way the OTP framework hosts a variety of modeling techniques, e.g., finite state machines.

Petri nets are a modeling technique especially suited for distributed systems. We introduce `gen_pnet`, a behavior for designing Erlang processes as Petri nets. We give a short introduction to Petri net semantics and demonstrate how Erlang applications can be modeled as Petri nets. Furthermore, we discuss two Erlang applications modeled and implemented as Petri nets. For both applications we introduce a Petri net model and discuss design challenges.

***CCS Concepts*** • **Software and its engineering** → *Development frameworks and environments*;

***Keywords*** OTP behavior, Petri nets, distributed systems

**ACM Reference Format:**
Jörgen Brandt and Wolfgang Reisig. 2018. Modeling Erlang Processes as Petri Nets. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3239332.3242767

## 1 Introduction

More and more software applications operate in a distributed setting. The fledging fields of service orientation [8] and microservices [3] opened up a new class of systems and come with their own design challenges. Partitioning a software application into independent services allows for a stricter modularization than is possible in closely coupled applications. This modularization unlocks a computational speedup through parallelization and the opportunity to use different software stacks provided they share an interface to communicate.

The Open Telecom Platform (OTP) [13] is a collection of libraries for Erlang. An OTP *behavior* is a design convention, frequently applied in Erlang libraries. It combines (i) a collection of functions acting as an application-independent scaffold for a process, controlling its life cycle and communication interface, and (ii) a collection of callback specifications which outline the application-specific parts of that process. Important examples are the `gen_server` or the `supervisor` behaviors.

Distributed systems introduce a new class of design challenges. A distributed application needs to cope with partial failure and protocols need to maintain invariants as well as liveness or termination properties which are seldom self-evident except in simplistic scenarios. In addition, distributed systems are rarely deterministic which adds a layer of complexity promoting transient failures which are hard to reproduce. Distribution complicates logging and debugging since no single component can hold a consistent view on the whole application. These challenges force us to design the components and protocols of a distributed system with extra care. The OTP framework offers several behaviors like `gen_fsm` to support the design of processes.

Petri nets are a modeling technique for distributed systems introduced in 1980 [5]. They have been applied to model biological systems [6], algorithms, or communication protocols. Petri nets are especially suited for modeling distributed software systems because they explicitly specify dependent and independent structures, which are central to the understanding of distribution. A Petri net makes progress in discrete transitional steps. Because Petri nets allow steps to occur independently transitional steps are, generally, unordered. Also, a transitional step assumes no knowledge about the system other than what directly enables it.

The independence of transitional steps in a Petri net mirrors the independence of events in distributed systems in which, generally, the order of events can be neither observed nor enforced. And the restriction of a transitional step to affect only a closed and predefined set of adjacent components mirrors the fragmentation of distributed systems in which, generally, the state of the system as a whole can be neither


*Erlang '18, September 29, 2018, St. Louis, MO, USA*

ACM ISBN 978-1-4503-5824-8/18/09…$15.00
https://doi.org/10.1145/3239332.3242767

observed nor enforced. Consequently, Petri nets are an ideal modeling framework for distributed systems.

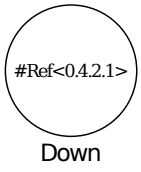
We introduce `gen_pnet`[1], an OTP behavior to model Erlang processes as Petri nets. `gen_pnet` allows specifying high-level interface nets. *High-level* means that transitions perform arbitrary operations on arbitrary data, i.e., Erlang data structures, as opposed to just black tokens. This is important since we need to model concrete software and not just an abstract, simplified version of it. *Interface nets* can interact with their environment. This is important because Petri net instances need to communicate with the outside world by sending and receiving messages and reacting to process failures. A `gen_pnet` instance appears to the outside as an ordinary Erlang process while its internal behavior is defined as a Petri net. This is similar to the way a `gen_fsm` appears to the outside as an ordinary Erlang process while its internal behavior is defined as a finite state machine.

We demonstrate the applicability of the `gen_pnet` behavior to two selected problems: a worker pool manager and a distributed programming language. With these use cases we show how Petri net instances communicate with other Erlang processes, how they handle partial system failure by translating messages and events to tokens in the net, and how they are composed to form large systems. We show that it is possible to exhaustively model complex applications in a concise, understandable way. Herein, we use the functional programming facilities, versatile data representations, and the pattern matching facilities of Erlang extensively.

The remainder of the paper is organized as follows: In Section 2 we provide an informal introduction to Petri nets. Section 3 describes the `gen_pnet` behavior and examplifies the implementation of a simple Petri net. In Section 4 we cover two applications based on `gen_pnet`. Section 5 discusses related work and Section 6 concludes the paper.

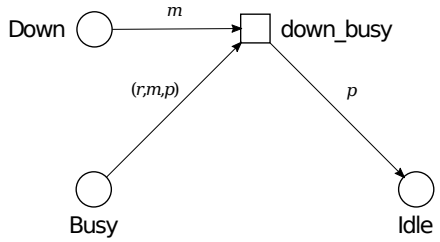
## 2 Petri Nets

In this section we give a self-contained but informal introduction to Petri nets. See Reisig 2013 [12] for a more thorough introduction to Petri nets. Petri nets describe distributed applications by distinguishing *passive* and *active* components. Passive components store an unordered collection of possibly identical values. A passive component is called a *place* and is represented by a circle. The values a place holds are written inside it. A value on the place of a Petri net is called a *token* and the entirety of all tokens in a Petri net is called the Petri net's *marking*. Below is a place with the name Down holding a monitor reference value.




[1] https://github.com/joergen7/gen_pnet

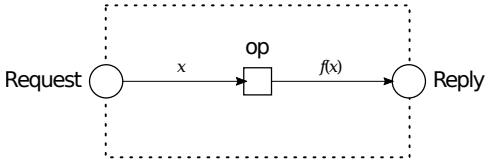
Active components are called *transitions*. Transitions and places are connected in a directed graph. Places with arcs going into a transition, are called the transition's *preset*. Places with incoming arcs from a transition are called the transition's *postset*. Transitions process data by consuming a token from each place in its preset and producing tokens on each place in its postset in an atomic step. Below is a transition with two places in its preset and one place in its postset. The net is unmarked.




By convention, we write place names upper-camel-case while transition names are written lower-snake-case. Each arc is annotated with a pattern. Only tokens that match the pattern annotation *enable* the transition. In addition, transitions can be annotated with *guards* specifying under which conditions the transition can fire.

When a transition *fires*, it consumes one token from each of its preset places. The tokens produced by the transition are a function of the input tokens. The binding of the pattern variables in the arc annotations is called the transition's *mode*. The pattern variable bindings must be consistent over all arcs in the preset of the transition in question.

Petri nets often represent open systems, i.e., systems that interact with their environment. We represent the system's *interface* to the environment with a dotted line. Places that are on the dotted line are considered *interface places*. An interface place with only outgoing arcs is considered an input place and tokens may spontaneously appear on that place. Conversely, an interface place with only incoming arcs is considered an output place and tokens may spontaneously disappear from that place. Here, we consider transitions on an interface or arcs crossing an interface are considered as malformed. Below, is an interface net with one input place Request and one output place Reply.




The Petri nets we discuss here, carrying arbitrary data, allowing arbitrary computation in transitions and interacting with their environment via an interface are called *high-level interface nets*. This kind of net is especially suited for modeling software because it is concrete, i.e., capable of specifying data and computation, and open, i.e., part of an environment instead of a closed system.

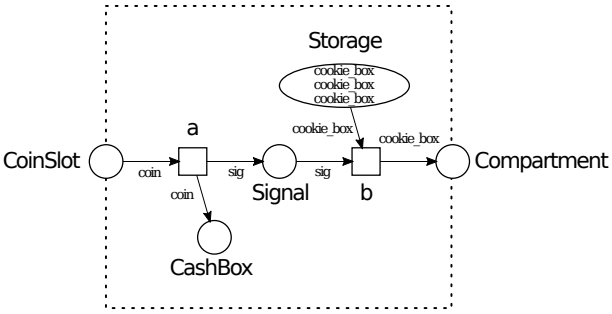



**Figure 1.** Petri net model of a cookie vending machine

## 3 Erlang Processes as Petri Nets

The `gen_pnet` OTP behavior allows to define high-level interface nets in Erlang. It extends the `gen_server` behavior to allow seamless integration into Erlang applications: A `gen_pnet` instance is a process which can send and receive messages, can be called, cast on, monitored, linked to, supervised, or handle hot code reloading.

We define Petri nets by creating a callback module that implements the `gen_pnet` behavior providing the following callback functions.

- **`place_lst/0`** returns the names of the places in the net.
- **`trsn_lst/0`** returns the names of the transitions in the net.
- **`init_marking/2`** returns the initial marking for a given place.
- **`preset/1`** returns the preset places of a given transition.
- **`is_enabled/2`** determines whether a given transition is enabled in a given mode.
- **`fire/3`** returns which tokens are produced on what places if a given transition is fired in a given mode that enables this transition.
- **`trigger/3`** allows to add a side effect to the generation of a token.

In addition, the `gen_pnet` behavior defines some more callback functions like `code_change/3` or `handle_info/2` which hand up the functionality of the `gen_server` behavior it extends.

### 3.1 Example: Cookie Vending Machine

We exemplify the implementation of a Petri net with `gen_pnet` by implementing the cookie vending machine depicted in Figure 1. For the sake of simplicity this cookie vending machine model provides neither a way to refill the storage nor to empty the cash box since both places are inaccessible via the interface.

Next, we define each callback function in turn. The `place_lst/0` function lets us define the names of all places in the net. Here, we define the net to have the five places in the cookie vending machine.

```
place_lst() ->
  ['CoinSlot', 'CashBox', 'Signal', 'Storage',
   'Compartment'].
```

The `trsn_lst/0` function lets us define the names of all transitions in the net. Here, we define the net to have the two transitions a and b.

```
trsn_lst() -> [a, b].
```

The `preset/1` function lets us define the preset places of a given transition. As its argument it takes the transition's name. Here, we define the preset of the transition a to be just the place CoinSlot while the transition b has the places Signal and Storage in its preset.

```
preset( a ) -> ['CoinSlot'];
preset( b ) -> ['Signal', 'Storage'].
```

The `init_marking/2` function lets us define the initial marking for a given place in the form of a token list. As arguments it takes a place name and a user info field. Here, we initialize the storage place with three `cookie_box` tokens. All other places are left empty.

```
init_marking( 'Storage', _UsrInfo ) ->
  [cookie_box, cookie_box, cookie_box];

init_marking( _Place, _UsrInfo ) ->
  [].
```

The `is_enabled/3` function is a predicate determining if a given transition is enabled in a given mode. As arguments it takes a transition name, a firing mode in the form of a hash map mapping place names to token lists, and a user info field. Here, we state that the transition a is enabled if it can consume a single `coin` from the CoinSlot place. Similarly, the transition b is enabled if it can consume a `sig` token from the Signal place and a `cookie_box` token from the Storage place. No other configuration can enable a transition. E.g., managing to get a `button` token on the CoinSlot place will not enable any transition.

```
is_enabled( a, #{ 'CoinSlot' := [coin] },
            _UsrInfo ) ->
  true;

is_enabled( b, #{ 'Signal'  := [sig],
                  'Storage' := [cookie_box] },
            _UsrInfo ) ->
  true;

is_enabled( _Trsn, _Mode, _UsrInfo ) ->
  false.
```

The `fire/3` function defines what tokens are produced when a transition fires in a given mode. Like `is_enabled/3` it takes as arguments a transition name, a firing mode, and a user info field. The `fire/3` function is called only on modes for which `is_enabled/2` returns `true`. The `fire/3` function is expected to return either a `{produce, ProduceMap}` tuple

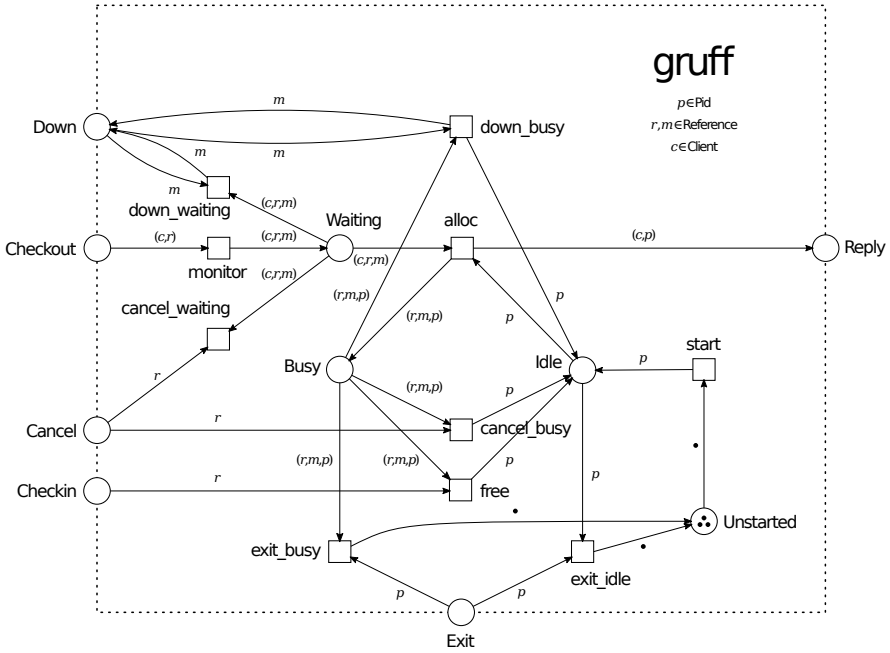



**Figure 2.** Petri net model of the gruff worker pool manager initialized with three workers

or the atom `abort`. If `abort` is returned, the firing is canceled, i.e., nothing is produced or consumed. Here, the firing of the transition a produces a `coin` token on the CashBox place and a `sig` token on the Signal place. Similarly, the firing of the transition b produces a `cookie_box` token on the Compartment place. We do not need to match in the function head the tokens to be consumed because the firing mode already uniquely identifies these tokens.

```
fire( a, _Mode, _UsrInfo ) ->
  {produce, #{ 'CashBox' => [coin],
               'Signal'  => [sig] }};

fire( b, _Mode, _UsrInfo ) ->
  {produce, #{ 'Compartment' => [cookie_box] }}.
```

The `trigger/3` function determines what happens when a token is produced on a given place. As arguments it takes a place name, the token about to be produced, and a user info field. The `trigger/3` function is expected to return either `pass` in which case the token is produced normally, or `drop` in which case the token is forgotten. Here, we simply let any token pass.

```
trigger( _Place, _Token, _UsrInfo ) -> pass.
```

## 4 Applications

In the following we discuss two applications of the `gen_pnet` behavior to demonstrate its applicability.

### 4.1 Gruff: A Worker Pool Manager

Gruff[2] is an adaptation of the worker pool manager poolboy demonstrating the applicability of `gen_pnet`. We chose poolboy because it is one of the most popular applications in the Erlang community, thus, providing a representative use case.

Poolboy manages access to a set of worker processes. A worker process typically performs computation. In this case, limiting access to worker processes reflects the scarcity of computational resources in a system. Another use case is the management of database connections which are costly to initialize. In this case poolboy restricts access to a limited number of database connections which are reused over time. Thus, poolboy is a generalization of a mutual exclusion algorithm [10] protecting access to a set of resources via a single gatekeeper.

Gruff implements this functionality in terms of a Petri net (see Figure 2). Gruff is not an exhaustive reimplementation of poolboy although its interface to the outside is identical.

A gruff instance receives check-out requests on the place Checkout in the form of a pair denoting the client process $c$ and a transaction reference $r$ that is a unique identifier of the session generated by the client. Gruff replies to check-out requests by generating a token consisting of the client $c$, the session reference $r$ and the associated worker process id $p$ on the Reply place.

When the worker pool manager receives a check-out request it creates a monitor so that a failing client can be

[2] https://github.com/joergen7/gruff

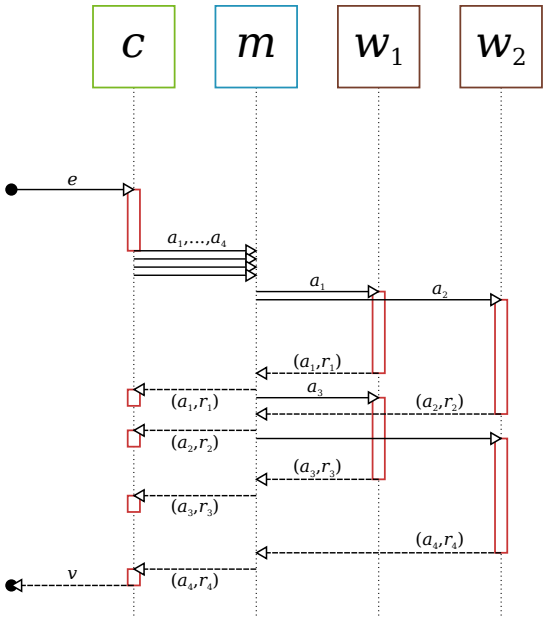



**Figure 3.** Sequence diagram of a distributed programming language. The system comprises a client $c$, a master $m$ and two workers $w_1$ and $w_2$. The expression $e$ is evaluated until a value $v$ results.

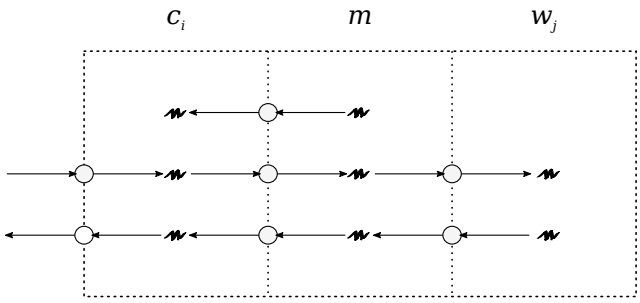



**Figure 4.** Composition of client, master, and worker nets with respective interface places

detected. Then, it attempts to allocate a worker process which succeeds if at least one idle worker is available. Next, a token appears on the Reply place notifying the client and the worker pool manager marks the allocated worker busy. When done, a client checks in the allocated worker. In this event, a token appears on the Checkin place and the worker pool manager marks the allocated worker idle again, which makes it available for other check-out requests.

In addition, a client can cancel a check-out request. Furthermore, if a client fails, all associated workers are marked idle. In contrast, if a worker fails, a new worker is spawned.

### 4.2 Cuneiform: A Distributed Language

Cuneiform [1, 2] is a distributed functional programming language which focuses on the integration of external tools and libraries. Cuneiform has been the original motivation to develop `gen_pnet`, which makes it a natural candidate for demonstrating its applicability. Cuneiform's distributed execution engine, the common execution environment (CRE)[3], is modeled and implemented as a Petri net using `gen_pnet`.

The CRE receives an expression $e$ which is evaluated and generates a set of independent function applications. For each application $a_i$ the result $r_i$ is computed by one of the workers $w_i$. Since the applications are independent they can be scheduled to distributed worker nodes. Upon learning about a computation result the interpreter makes progress, further evaluating the expression until a value results. The value is returned to the user and interpretation terminates. Figure 3 shows a sequence diagram illustrating the message exchanged to evaluate an expression consisting of four independent applications using two distributed workers. Figure 4 sketches how client $c$, master $m$ and workers $w_i$ are composed as a Petri net.

The CRE master $m$ is a combination of a cache and a scheduler (see Figure 5). The left part of net represents the demand circuit which is used to limit the rate at which clients can send applications to the CRE master. The middle part represents the cache which consists of a guard allowing only fresh applications to pass to the scheduler. All other applications are served through the lookup transition as soon as the result becomes available. The right part represents the scheduler circuit. Process ids of idle workers appear on the WorkerPool place. When an application is scheduled an application-worker process id pair is moved to the BusyWorker place. On receiving an application-result pair from a worker the worker process id is returned to the WorkerPool place.

The CRE master must deal with the dynamic addition and failure of worker nodes. Should a worker fail while it is still processing an application, it reschedules the application and removes the worker from the BusyWorker place. Workers can sign on to the CRE master at any time generating a token on the AddWorker place.

## 5 Related Work

CPNTools [9] is an open source tool suite based on colored Petri nets for editing, simulating, and analysing Petri nets. It offers structure-based analysis tools and has a focus on usability and graphical interfaces. It allows Petri nets to operate on arbitrary data and to perform computations, specified in Standard ML. CPNTools can generate Erlang code [7]. The generated code is used as an application scaffold which is manually modified [4].

Haskell-corored Petri nets are a mapping of Colored Petri nets to Haskell by creating embeddings [11]. This is similar to the approach of `gen_pnet`. `gen_pnet` can be seen as a generic embedding that is configured with a Petri net structure.

[3] https://github.com/joergen7/cre

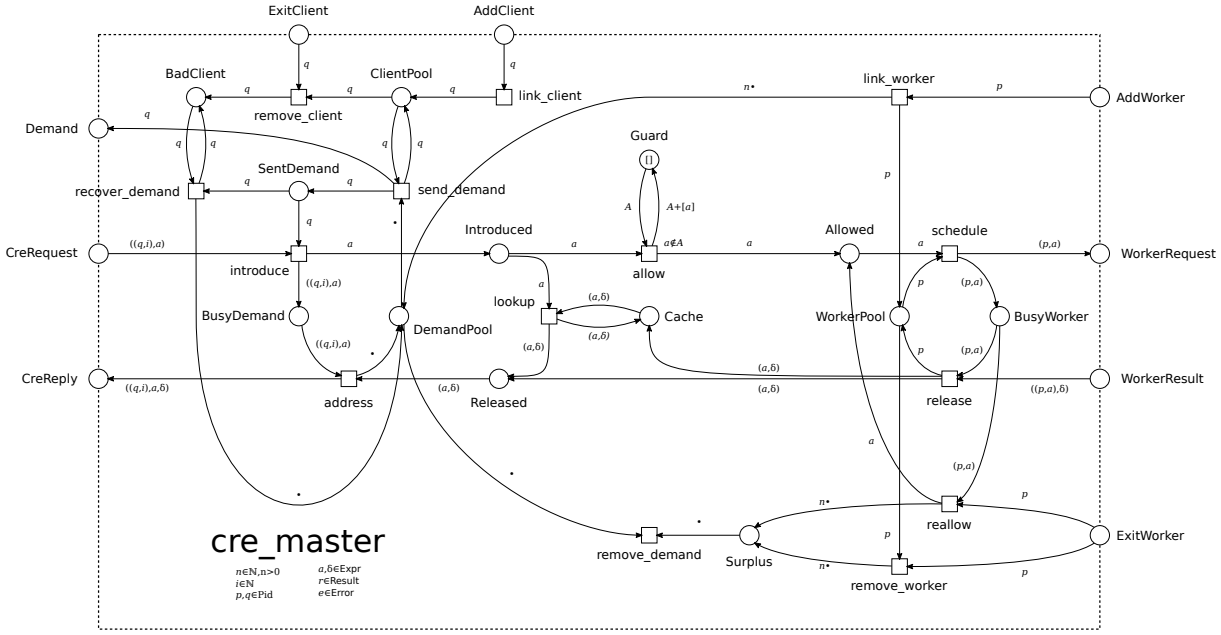



**Figure 5.** Petri net model of CRE master

gen_pn[4] is a different Erlang library allowing to define Petri nets in Erlang in an OTP-like way. The main difference is that gen_pn focuses on the definition of elementary Petri nets instead of high-level nets. The idea of modeling and implementing software using Petri nets is not new. What sets gen_pnet apart is the tapping of high-level Petri nets in Erlang while playing well with the way distributed applications are composed in the OTP framework.

## 6 Conclusion

We have introduced gen_pnet, a behavior for defining Erlang processes as Petri nets. The behavior lets users specify a Petri net structure and an initial marking via callback functions. Basing the behavior on a gen_server allows to compose gen_pnet with other Petri net instances or Erlang processes to generate versatile distributed applications. We have demonstrated the applicability of Petri nets by implementing a worker pool manager and a distributed programming language.

## References


[1] Jörgen Brandt, Marc Bux, and Ulf Leser. 2015. A functional language for large scale scientific data analysis. In *EDBT/ICDT 2015 Joint Conference*. 7.

[2] Jörgen Brandt, Wolfgang Reisig, and Ulf Leser. 2017. Computation semantics of the functional scientific workflow language Cuneiform. *Journal of Functional Programming* 27 (2017).

[3] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. *Microservices: Yesterday, Today, and Tomorrow.* Springer International Publishing, Cham, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

[4] Kristian Leth Espensen, Mads Keblov Kjeldsen, Lars Michael Kristensen, and Michael Westergaard. 2009. Towards Automatic Code-generation from Process-partitioned Coloured Petri Nets. In *Proc. of 10th CPN Workshop*. 41–60.

[5] H. J. Genrich, K. Lautenbach, and P. S. Thiagarajan. 1980. Elements of general net theory. In *Net Theory and Applications*, Wilfried Brauer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–163.

[6] Monika Heiner, David Gilbert, and Robin Donaldson. 2008. Petri Nets for Systems and Synthetic Biology. In *Formal Methods for Computational Systems Biology*, Marco Bernardo, Pierpaolo Degano, and Gianluigi Zavattaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–264.

[7] Lars Michael Kristensen and Michael Westergaard. 2010. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Formal Methods for Industrial Critical Systems*, Stefan Kowalewski and Marco Roveri (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–230.

[8] R. Perrey and M. Lycett. 2003. Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.* 116–119. https://doi.org/10.1109/SAINTW.2003.1210138

[9] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. 2003. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Applications and Theory of Petri Nets 2003*, Wil M. P. van der Aalst and Eike Best (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 450–462.

[10] M. Raynal. [n. d.]. Algorithms for mutual exclusion. ([n. d.]).

[11] Claus Reinke. 2000. Haskell-Coloured Petri Nets. In *Implementation of Functional Languages*, Pieter Koopman and Chris Clack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 165–180.

[12] Wolfgang Reisig. 2013. *Understanding petri nets: modeling techniques, analysis methods, case studies.* Springer.

[13] Seved Torstendahl. 1997. Open telecom platform. *Ericsson Review(English Edition)* 74, 1 (1997), 14–23.


[4] https://github.com/aabs/gen_pn